Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Операционные системы

Отчет по лабораторной работе №5

Управление процессами и потоками в Windows

Работу выполнил:

Иванов А.А.

Группа: 43501/4

Преподаватель:

Малышев И.А.

Санкт-Петербург

2017

# Цель работы

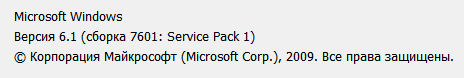
1. Исследовать возможность создания нескольких процессов с базовым потоком и нескольких потоков в рамках одного процесса.
2. Проанализировать выделение процессорного ресурса потокам с изменением их приоритетов.

# Программа работы

1. Программа после запуска должна создать новый процесс, с помощью функции CreateProcess. В новом процессе необходимо запустить любое приложение
   1. Программа, получает имя конфигурационного файла из командной строки, открывает конфигурационный файл, читает строки и создает для запуска каждой команды отдельный процесс.
   2. Пусть программа получает имя конфигурационного файла из командной строки, открывает его с помощью fopen(), читает построчно функцией fgets(). После прочтения каждой строки, если она не пуста, создается процесс, в командную строку которого пишется прочитанная строка. Если создать процесс не удалось, программа пробует читать конфигурационный файл дальше.
2. Программа должна создавать два потока, выводящих в бесконечном цикле «1» и «2» соответственно. После создания дополнительных потоков, поток-родитель завершается. В данной программе после создания потоков, главный поток завершается, способов завершения может быть два: с помощью return и с помощью вызова функции ExitThread.
   1. Программа должна получать 2 параметра – количество создаваемых потоков и время жизни всего приложения. С интервалом в 1 сек каждый рабочий поток выводит о себе информацию и отслеживает состояние переменной, которая устанавливается в заданное значение по истечении времени жизни процесса.
3. Подготовить программу, в которой у каждого из потоков свой приоритет отличный от других. Все они выполняют одинаковую работу, например, увеличивают каждый свой счетчик. Накопленное значение счетчика, таким образом, отражает относительное суммарное время выполнения потока. Предполагаем, так как приоритеты различны, то и время, отведенное на работу потокам различно (квант времени, выделяемый потокам, одинаков).
   1. Усложним задачу и дополним ее возможностью управлять классом приоритетов процесса. Код программы несколько изменим для получения более наглядного вывода результатов. Пусть программа по-прежнему создает 7 дополнительных потоков, со всеми возможными вариантами приоритета. Теперь в начале работы можно изменить класс приоритета процесса в целом. Каждый рабочий процесс выполняет увеличение связанного с ним счетчика (здесь типа int).
   2. Анализ поведения системных функций динамического управления приоритетами 1 процессов и потоков. С помощью программы определим, назначается ли динамическое изменение приоритетов по умолчанию, на все ли потоки воздействует функция SetProcessPriorityBoost(), возможно ли разрешение отдельному потоку в процессе динамически изменять приоритет, если для процесса это запрещено.
4. Исследуйте результаты работы программы 3.1 и 3.2 в зависимости от того, какой приоритет назначается базовому потоку: аbove\_normal, idle\_priority class, high priority class, normal priority class и др.; своими экспериментальными данными заполните таблицу, с точным указанием для какой ОС и на каком отладочном комплексе проводились измерения.
5. Модифицируйте программу 3.2 для заполнения таблицы 2 текущими данными вашего эксперимента. Сделайте выводы.
6. С помощью утилит CPU Stress, позволяющих нагружать систему, и утилиты мониторинга ProcessExplorer() (или иных утилит) зафиксируйте динамическое изменение приоритетов, приведите результаты в отчете.
7. Создайте программу, демонстрирующую возможность наследования: дескриптора порождающего процесса, дескрипторов открытых файлов. Для выполнения этого задания следует учесть, что по умолчанию наследование в Windows отключено и для возможности наследования, необходимо: разрешить процессу-потомку наследовать дескрипторы, сделать дескрипторы наследуемыми

# Ход работы

Версия используемой ОС:

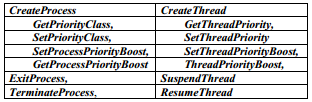


# 1. Создание процессов

Создание системных приложений в ОС Windows осуществляется с применением Application Programming Interface (интерфейса программирования приложений). Перечислим основные функции управления процессами и потоками: создания, завершения, управления приоритетами.

Функции управления процессами и потоками:

*Рисунок 1. Функции управления процессами и потоками*



Рассмотрим создание одного процесса другим посредством функции WinAPI CreateProcess. При создании исполнительная система выполняет работу по организации окружения (среды исполнения процесса) и предоставлению необходимых ему ресурсов. Она выделяет новое адресное пространство и иные ресурсы для процесса, а также создает для него новый базовый поток. Когда новый процесс будет создан, старый процесс будет продолжать исполняться, используя старое адресное пространство, а новый будет выполняться в новом адресном пространстве с новым базовым потоком. Существует много различных опций для создания процесса, поэтому функция CreateProcess( ) имеет порядка десяти параметров, причем некоторые из них достаточно сложные и информационно емкие. После того, как исполнительная система создала новый процесс, она возвращает его описатель, а также описатель его базового потока.

## 1.1. Создание процесса для запуска приложения

Программа после запуска должна создать новый процесс, с помощью функции CreateProcess.

В новом процессе необходимо запустить любое приложение (например, notepad.exe или calc.exe). Для контроля можно вывести идентификаторы созданного процесса и потока, а затем завершить основную программу.

Используем функцию CreateProcess, которая создает новый процесс: выделяет новое адресное пространство и иные ресурсы процессора, создает базовый поток. Когда новый процесс будет создан, старый процесс будет продолжать исполняться, используя старое адресное пространство, а новый будет выполняться в новом адресном пространстве с новым базовым потоком. После того, как исполнительная система создала новый процесс, она возвращает его описатель, а также описатель его базового потока. Синтаксис команды CreateProcess:

|  |
| --- |
| BOOL CreateProcess(  LPCTSTR lpApplicationName, // имя исполняемого модуля  LPTSTR lpCommandLine, // командная строка  LPSECURITY\_ATTRIBUTES lpProcessAttributes, // атрибуты безопасности процесса  LPSECURITY\_ATTRIBUTES lpThreadAttributes, // атрибуты безопасности потока  BOOL bInheritHandeles, // флаг наследования описателя  DWORD dwCreationFlags, // флаги создания  LPVOID lpEnvironment, // новый блок окружения  LPCTSTR lpCurrentDirectory, // имя текущей директории  LPSTARTUPINFO lpStartupInfo, // STARTUPINFO  LPPROCESS\_INFORMATION lpProcessInformation // PROCESS\_INFORMATION  ) |

Десять параметров CreateProcess() обеспечивают большую гибкость при использовании программистом, в простейшем случае для многих параметров можно использовать значения по умолчанию.

#### Параметры lpApplicationName и lpCommandLine

Используются вместе для указания исполняемой программы и аргументов командной строки.

#### Параметры lpProcessAttributes, lpThreadAttributes и bInheritHandeles

Первые два – указатели на атрибуты безопасности для процесса и потока соответственно, последний параметр – флаг наследования (наследуются ли файловые дескрипторы и т.д.).

#### Параметр DwCreationFlags

Может объединять в себе несколько флаговых значений, включая следующие:

* CREATE\_SUSPENDED — указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функция ResumeThread;
* DETACHED\_PROCESS и CREATE\_NEW\_CONSOLE — взаимоисключающие значения, которые не должны устанавливаться оба одновременно. Первый флаг означает создание нового процесса, у которого консоль отсутствует, а второй — процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса;
* CREATE\_NEW\_PROCESS\_GROUP — указывает на то, что создаваемый процесс является корневым для новой группы процессов. Если все процессы, принадлежащие данной группе, разделяют общую консоль, то все они будут получать управляющие сигналы консоли (Ctrl-C или Ctrl-break);

В качестве флагов так же могут быть указаны приоритеты: HIGH\_PRIORITY\_CLASS, IDLE\_ PRIORITY\_CLASS, NORMAL\_PRIORITY\_CLASS или REALTIME\_PRIORITY\_CLASS. Значение по умолчанию - NORMAL\_PRIORITY\_CLASS, но если порождающий процесс имеет приоритет IDLE\_PRIORITY\_CLASS, то и процесс-потомок также будет иметь приоритет IDLE\_PRIORITY\_CLASS.

#### Параметр lpEnvironment

Используется для передачи нового блока переменных окружения порожденному процессупотомку. Если NULL, то потомок использует то же окружение, что и родитель. Если не NULL, то lpEnvironment должен указывать на массив строк, каждая name=value.

#### Параметр lpCurrentDirectory

Определяет полное путевое имя директории, в которой потомок будет выполняться. Если использовать NULL, то потомок будет использовать директорию родителя.

#### Параметр lpStartupInfo

lpStartupInfo – это указатель на структуру STARTUPINFO, которая устанавливает оконный режим терминала, рабочий стол, стандартные дескрипторы и внешний вид главного окна для нового процесса.

|  |
| --- |
| typedef struct\_STARTUPINFO {  DWORD cb; // длина структуры (обязательно инициализируем)  LPTSTR lpReserved;  LPTSTR lpDesktop;  LPTSTR lpTitle;  55  DWORD dwX;  DWORD dwY;  DWORD dwXSize;  DWORD dwSizeY;  DWORD dwXCountChars;  DWORD dwYCountChars;  DWORD dwFillAttribute;  DWORD dwFlags;  DWORD dwShowWindow;  WORD cbReserved2;  LPBYTE lpReserved2;  HANDLE hStdInput;  HANDLE hStdOutput;  HANDLE hStdError;  }; |

#### Параметр lpProcessInformation

Указатель на структуру PROCESS\_INFORMATION, которая принимает идентифицирующую информацию о новом процессе.

|  |
| --- |
| Typedef struct\_PROCESS\_INFORMATION {  HANDLE hProcess; //описатель нового процесса  HANDLE hThread; //описатель потока нового процесса  DWORD dwProcessId; //глобальный идентификатор созданного процесса  DWORD dwThreadId; //глобальный идентификатор потока созданного процесса  }; |

CreateProcess() отводит место под объекты процесс и поток и возвращает значения их описателей (индексы в таблице) в структуре PROCESS\_INFORMATION.

Освободить выделенное место можно вызовом CloseHandle. При этом выполнение этого вызова не обязательно приведет к завершению процесса (только исчезнет ссылка на объект внутри вызвавшего процесса).

CreateProcess() возвращает ноль, если создание процесса прошло успешно.

Таким образом, используя функцию CreateProcess создаем новый процесс для запуска приложения. Для контроля работы функции выводим на консоль идентификаторы созданного процесса и потока. Программа task1.cpp:

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  #include <string.h>  #include <iostream>  #include<TCHAR.H>  using namespace std;  #define \_CRT\_SECURE\_NO\_WARNINGS  #pragma warning(disable : 4996)  int main(int argc, char\*\* argv[]) {  printf("Program started\n");  TCHAR \*path = new TCHAR[45];  strcpy(path, \_T("C:\\Windows\\System32\\calc.exe"));  LPTSTR LpCommandLine = path; // string wt command in universal form  STARTUPINFO startupInfo; // pointer to structure  PROCESS\_INFORMATION processInfo; // info about process  ZeroMemory(&startupInfo, sizeof(STARTUPINFO));  startupInfo.cb = sizeof(startupInfo);  // creating process  if (!CreateProcess(NULL, LpCommandLine, NULL, NULL, false, HIGH\_PRIORITY\_CLASS |  CREATE\_NEW\_CONSOLE, NULL, NULL, &startupInfo, &processInfo)) {  cout << "Error creating process: " << GetLastError() << endl;  return -1;  }  else {  cout << "New process Handle: " << processInfo.dwProcessId <<  " Handle of thread: " << processInfo.dwThreadId << endl <<  "Successfully created new process!" << endl;  }  CloseHandle(processInfo.hThread); // закрытие нити  CloseHandle(processInfo.hProcess); // закрытие процесса  cout << "Program finished" << endl;  getchar();  return 0;  } |

Параметр lpApplicationName установлен NULL. В этом случае, имя модуля должно быть в строке lpCommandLine.

В данном случае будем использовать значения по умолчанию для атрибутов безопасности процесса и потока –– NULL (параметры lpProcessAttributes и lpThreadAttributes ) и FALSE для флага наследования (bInheritHandles).

Для создания нового процесса (child) с высоким приоритетом в его собственном окне используем — HIGH\_PRIORITY\_CLASS | CREATE\_NEW\_CONSOLE.

Параметр lpEnvironment используется для передачи нового блока переменных окружения порожденному процессу-потомку (child). Если указано NULL —- то потомок использует то же окружение, что и родитель.

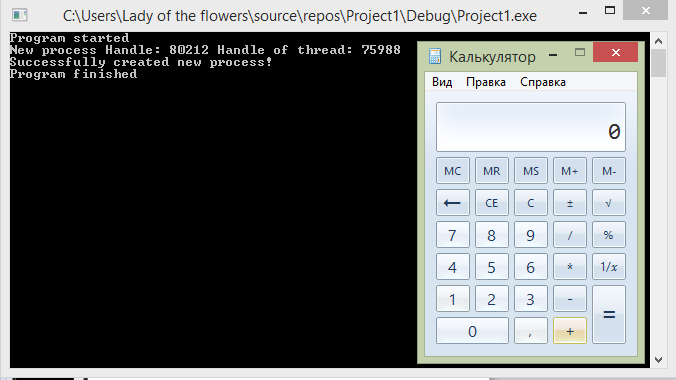
Параметр lpCurrentDirectory установлен в (NULL). Это означает, что новый процесс создается с тем же самым текущим диском и каталогом, что и вызывающий процесс.

В структуре startupInfo устанавливаются оконный режим терминала, рабочий стол, стандартные дескрипторы и внешний вид главного окна для нового процесса.

В структуре processInfo хранятся: описатель вновь созданного процесса (hProcess), описатель его базового потока (hThread), глобальный идентификатор процесса (dwProcessId), глобальный идентификатор потока (dwThreadId).

Программа работает верно: создает новый процесс, выводит идентификаторы нового процесса и его базового потока, после чего завершается. О создании нового процесса можно судить, по созданию нового окна с вызываемой программой. Результат работы программы приведен на рисунке 2.

*Рис. 2: Результат выполнения Project1.cpp*



## 1.2. Создание процессов при работе с конфигурационным файлом

Программа, получает имя конфигурационного файла из командной строки, открывает конфигурационный файл, читает строки и создает для запуска каждой команды отдельный процесс.

С целью упрощения кода сначала имя командного файла зададим прямо в программе и представим текст программы без обработки возможных ошибок. Создаем конфигурационный файл с именем "temp.txt" при помощи текстового редактора(например, notepad) и располагаем его в корневом каталоге диска С. Далее записываем в файл следующие строки:

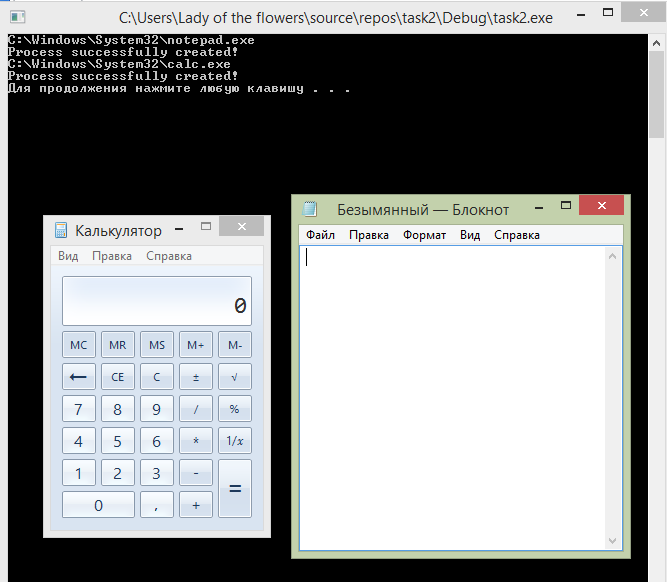
|  |
| --- |
| C:\Windows\System32\notepad.exe C:\Windows\System32\calc.exe |

Программа task2.cpp:

|  |
| --- |
| #include <stdio.h>  #include <windows.h>  #include <assert.h>  #include <iostream>  #define MAX\_LEN 200  using namespace std;  int main( int argc, char\* argv[] ) {  const char\* frd = "C:\\7\\OS\\5\\listings\\temp.txt";  FILE \*f = fopen( frd,"r" );  if( f == NULL ) {  cout << "Coudn't open file" << endl;  system( "pause" );  return 1;  }  for( int i = 0; i < 2; i++ ) {  char\* execString = (char\*)calloc( MAX\_LEN, sizeof(char) ); // memory allocation  fgets(execString, MAX\_LEN, f); // reading string from file  execString[strlen(execString) - 1] = '\0';  STARTUPINFO startupInfo;  ZeroMemory( &startupInfo, sizeof(STARTUPINFO) );  startupInfo.cb = sizeof( startupInfo );  PROCESS\_INFORMATION processInfo;  cout << execString << endl;  if( !CreateProcess( NULL, execString, NULL, NULL, false, 0, NULL, NULL, &startupInfo, &processInfo ) ) {  cout << "Error creating process: " << GetLastError() << endl;  return -1;  } else  cout << "Process successfully created!" << endl;  free( execString );  CloseHandle( processInfo.hThread );  CloseHandle( processInfo.hProcess );  }  return 0;  } |

Программа работает верно: открывает файл конфигурации, построчно считывает его, создает новый процесс для каждой строки и завершается. О создании новых процессов можно судить, по созданию новых окон с вызываемыми программами.

*Рис.3: Результат выполнения task2.cpp*



## 1.3. Доработаем программу

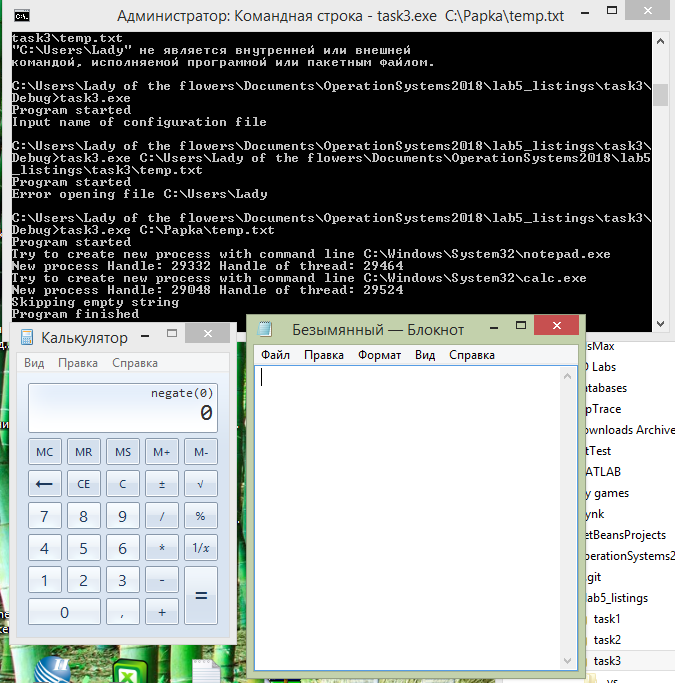
Пусть программа получает имя конфигурационного файла из командной строки, открывает его с помощью fopen(), читает построчно функцией fgets(). После прочтения каждой строки, если она не пуста, создается процесс, в командную строку которого пишется прочитанная строка. Если создать процесс не удалось, программа пробует читать конфигурационный файл дальше.

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include <stdio.h>  #include <windows.h>  #include <assert.h>  #include <iostream>  #define DEF\_BUFLEN 100  using namespace std;  int main( int argc, char\* argv[] ) {  cout << "Program started" << endl;  if (argc < 2) {  cout << "Input name of configuration file" << endl;  exit(999);  }  const char\* frd = argv[1];  FILE \*f = fopen( frd, "r" ); // open configuration file (reading)  if( f == NULL ) {  cout << "Error opening file " << argv[1] << endl;  exit(1000);  }  char commandLine[DEF\_BUFLEN]; // buffer for reading string  STARTUPINFO StartupInfo;  PROCESS\_INFORMATION ProcessInformation;  while( !feof(f) ) {  ZeroMemory( commandLine, DEF\_BUFLEN );  fgets( commandLine, DEF\_BUFLEN, f );  if( strlen(commandLine) <= 1 ) {  cout << "Skipping empty string" << endl;  continue;  }  commandLine[strlen(commandLine) - 1] = '\0';  ZeroMemory( &StartupInfo, sizeof(STARTUPINFO) );  StartupInfo.cb = sizeof( STARTUPINFO );  cout << "Try to create new process with command line " << commandLine << endl;  if( !CreateProcess(NULL, commandLine, NULL, NULL, false, 0, NULL, NULL, &StartupInfo, &ProcessInformation)){  cout << "Can't create new process. Error is: " << GetLastError() << ". Continue with nextline" << endl;  continue;  }  cout << "New process Handle: " << ProcessInformation.dwProcessId <<  " Handle of thread: " << ProcessInformation.dwThreadId << endl;  CloseHandle( ProcessInformation.hThread );  CloseHandle( ProcessInformation.hProcess );  }  fclose( f );  cout << "Program finished" << endl;  getchar();  return 0;  } |

Отметим, что вывод процессы производят здесь без синхронизации. Процесс-родитель не дожидается создания процессов-потомков, а им после создания необходимо разбирать строку аргументов. Когда передается один неверный аргумент, процесс не создается (т.к. сразу проверяется наличие исполняемого файла с указанным именем). «Error 2» соответствует ошибка «Файл не найден». При передаче в commandLine процесса строки из нескольких слов процесс все равно создается (даже если аргументы не верны), но быстро

завершается. При этом созданные процессы связаны с консолью процесса-родителя (аргумент DwCreationFlags в вызове CreateProcess равен 0)

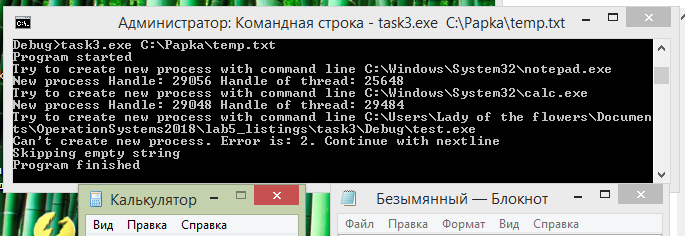
*Рис.4: Результат выполнения task3.cpp*



Результат работы соответствует ожиданиям. Изменим содержимое файла temp.txt:

|  |
| --- |
| C:\Windows\System32\notepad.exe  C:\Windows\System32\calc.exe  C:\Users\Lady of the flowers\Documents\OperationSystems2018\lab5\_listings\task3\Debug\test.exe |

*Рис.4: Результат выполнения task3.cpp с другими исходными данными*



Разберём результат работы программы построчно:

* Создается процесс ’C:\Windows\System32\notepad.exe’. Данный процесс создается без ошибок, номер процесс 29056, номер потока 29464.
* Создается процесс ’C:\Windows\System32\calc.exe’. Данный процесс создается без ошибок, номер процесс 29048, номер потока 29524.
* Пытаемся создаться процесс C:\Users\Lady of the flowers\Documents\OperationSystems2018\lab5\_listings\task3\Debug\test.exe
* Но такого .exe в данном каталоге не существует, запускать нечего. Создать процесс не получается, о чем и сообщает программа: «Can’t create new process. Error is: 2».
* Затем идет пустая строка, после чего программа завершает свою работу.

# 2. Создание потоков

Создание потоков производится посредством функции WinAPI CreateThread. Синтаксис команды CreateThread:

|  |
| --- |
| HANDLE CreateThread(  LPSECURITY\_ATTRIBUTES lpsa, //дескриптор защиты  DWORD dwStackSize, // начальный размер стека  LPTHREAD\_START\_ROUTINE lpStartAddr, //функция потока  LPVOID lpThreadParm, //параметр потока  DWORD dwCreationFlags, //опции создания  LPDWORD lpThreadId //идентификатор потока  ) |

Рассмотрим подробнее параметры функции:

* lpsa — указатель на структуру с атрибутами защиты.
* dwStackSize — размер стека нового потока в байтах. Значению 0 этого параметра соответствует размер стека по умолчанию, равный размеру стека основного потока.
* lpStartAddr — указатель на функцию (принадлежащую контексту процесса), которая должна выполняться. Эта функция принимает единственный аргумент в виде указателя и возвращает 32-битовый код завершения. Этот аргумент может интерпретироваться потоком либо как переменная типа DWORD, либо как указатель.  
  Функция потока (ThreadFunc) имеет следующую сигнатуру:

|  |
| --- |
| DWORD WINAPI ThreadFunc(LPVOID) |

* lpThreadParm — параметр главной функции потока.
* dwCreationFlags — если значение этого параметра установлено равным 0, то поток запускается сразу же после вызова функции CreateThread. Установка значения CREATE\_SUSPENDED приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности путем вызова функции ResumeThread.
* lpThreadId — указатель на переменную типа DWORD, которая получает идентификатор нового потока. Если NULL, то идентификатор не возвращается.

Если функция выполнилась успешно, то вернется описатель потока, если нет, вернется NULL.

# Управление потоками. Примеры применения функций создания потоков

## 2.1. Создание нескольких потоков

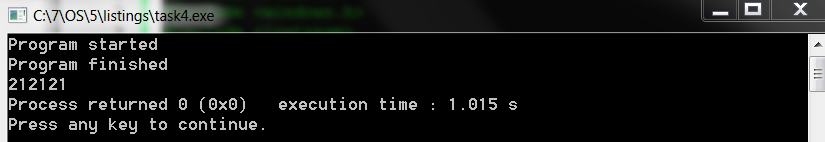
Задание. Программа должна создавать два потока, выводящих в бесконечном цикле «1» и «2» соответственно. После создания дополнительных потоков, поток-родитель завершается.

В данной программе task4.cpp после создания потоков, главный поток завершается, способов завершения может быть два: с помощью return и с помощью вызова функции ExitThread.

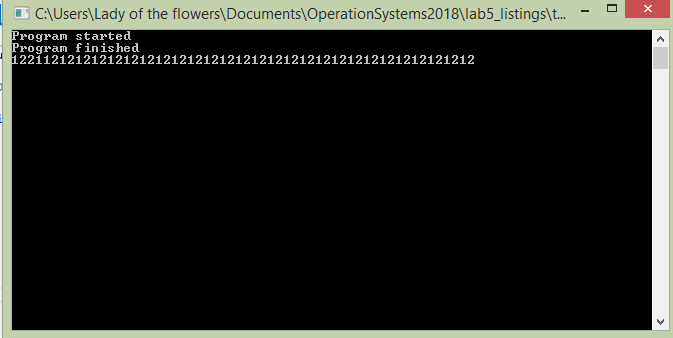
|  |
| --- |
| #include <stdio.h>  #include <windows.h>  #include <iostream>  using namespace std;  DWORD WINAPI threadHandler( LPVOID );  int main( int argc, char\* argv[] ) {  cout << "Program started" << endl;  HANDLE t;  int number = 1;  t = CreateThread( NULL, 0, threadHandler, (LPVOID)number, 0, NULL );  CloseHandle( t );  number = 2;  t = CreateThread( NULL, 0, threadHandler, (LPVOID)number, 0, NULL );  CloseHandle( t );  //ExitThread( 0 ); // 2 var  cout << "Program finished" << endl;  Sleep( 1000 );  return 0; // 1 var  }  DWORD WINAPI threadHandler( LPVOID param ) {  int number = (int)param;  for (;;) {  Sleep( 300 );  cout << number;  fflush( stdout );  }  return 0;  } |

Результаты выполнения программы:

*Рис.5: Результат выполнения task4.cpp с return*



*Рис.6: Результат выполнения task4.cpp с ExitThread()*



Очевидно, что в первом случае произошло завершение всего процесса (возврат из функции main), во втором – завершился только главный поток, а процесс – нет.

Аналогичные результаты будут, если основной поток после создания потомков выполнит функцию Sleep() с неопределенным временем ожидания.

Функция Sleep() позволяет потоку отказаться от использования процессора и перейти из состояния выполнения в состояние ожидания, которое будет длиться в течение заданного промежутка времени. Например, выполнение задачи потоком может продолжаться в течение некоторого периода времени, после чего поток приостанавливается. По истечении периода ожидания планировщик вновь переводит поток в состояние готовности.

|  |
| --- |
| VOID Sleep(DWORD dwMilliseconds) |

Длительность интервала ожидания указывается в милисекундах, и одним из ее возможных значений является INFINITE, что соответствует бесконечному периоду ожидания, при котором выполнение приостанавливается на неопределенное время. Значению 0 соответствует отказ потока от оставшейся части отведенного ей временного промежутка (если готовых потоков нет, то будет вызван этот же поток).

Использование Sleep()для синхронизации является очень неудачным решением, т.к. данная функция говорит только то, что по завершению таймаута поток/процесс перейдет в состояние «готов к выполнению». Порядок передачи управления определяет планировщик, поэтому предсказать очередность выполнения потоков не представляется возможным.

## 2.2. Создание программы, в которой время жизни процесса и порождаемых в нем потоков задается как параметр.

Задание. Программа должна получать 2 параметра – количество создаваемых потоков и время жизни всего приложения. С интервалом в 1 сек каждый рабочий поток выводит о себе информацию и отслеживает состояние переменной, которая устанавливается в заданное значение по истечении времени жизни процесса.

Возможны различные варианты решения данной задачи. Например, для подсчета времени можно использовать поток-координатор, вычисляющий момент завершения периода жизни с помощью функции getTickCount(), (сравнивая разницу текущего и стартового времени с заданным периодом жизни) или с помощью функции получения системного времени GetSystemTime(&now). Другой способ (более рациональный) – использование таймера ожидания (Waitable Timer).

Рассмотрим вариант на основе таймера ожидания.

Таймеры ожидания(waitable timers) – это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Чтобы создать ожидаемый таймер, достаточно вызвать функцию CreateWaitableTimer(). Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, необходимо вызвать функцию SetWaitableTimer().

|  |
| --- |
| HANDLE CreateWaitableTimer(  PSECURITY\_ATTRIBUTES psa,  BOOL fManualReset,  PCTSTR pszName) |

PSECURITY\_ATTRIBUTES psa – атрибуты безопасности (аналогичны как и для вызовов createThread, createProcess);

BOOL fManualReset – определяет тип ожидаемого таймера (со сбросом вручную или автосбросом). Когда освобождается таймер со сбросом вручную, то возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом – лишь один из потоков.

PCTSTR pszName – имя таймера, по которому можно получить его описатель (может быть равно NULL).

Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, необходимо вызвать функцию SetWaitableTimer.

|  |
| --- |
| BOOL SetWaitableTimer(  HANDLE hTimer,  const LARGE\_INTEGER \*pDueTime,  LONG lPeriod,  PTIMERAPCROUTINE pfnCompletionRoutine,  PVOID pvArgToCotnpletionRoutine,  BOOL fResume) |

HANDLE hTimer – описатель таймера;

const LARGE\_INTEGER \*pDueTime – время первого срабатывания таймера ( используется специальная структура FILETIME);

LONG lPeriod – период повторений срабатывания таймера (если равен 0, то сработает 1 раз; считается в миллисекундах);

BOOL fResume – позволяет вывести компьютер из режима сна, когда таймер срабатывает (иначе – таймер перейдет в свободное состояние, но ожидавшие его потоки не получат процессорное время, пока компьютер не выйдет из режима сна).

Остальные параметры берутся по-умолчанию.

Поток – координатор получает в качестве аргумента созданный таймер и запускает функцию WaitForSingleObject, которая возвращает управление, если объект освободился.

|  |
| --- |
| DWORD WaitForSingleObject( HANDLE hObject, DWORD dwMilliseconds) |

HANDLE hObject – описатель объекта;

DWORD dwMilliseconds – сколько времени в миллисекундах мы готовы ждать освобождения (если используется константа INFINITE, то ожидать будем бесконечно).

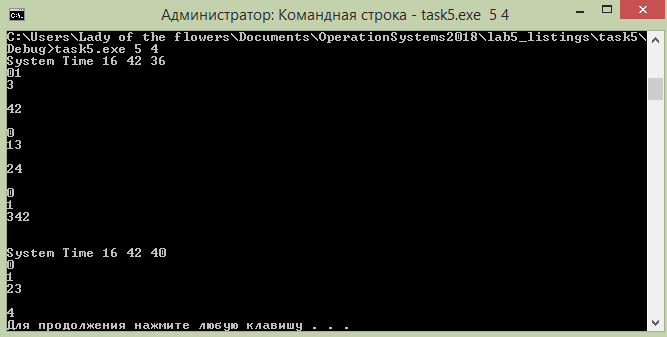
Чтобы посмотреть, что возвращает функция, можно использовать GetLastError:  
WAIT\_OBJECT\_O – если послано уведомление;  
или WAIT\_TIMEOUT – истек тайм-аут.

Программа task5.cpp:

|  |
| --- |
| #include <stdio.h>  #include <windows.h>  #include <iostream>  using namespace std;  DWORD WINAPI Thread1(LPVOID);  int stop;  struct params {  int num;  bool\* runflg;  };  int main( int argc, char\* argv[] ) {  SYSTEMTIME now;  int thrds;  if( argc < 3 )  thrds = 2; // number of threads  else  thrds = atoi( argv[1] );  if( argc < 3 )  stop = 5000; // time to stop  else  stop = atoi( argv[2] );  DWORD targetThreadId;  bool runFlag = true;  \_\_int64 end\_time;  LARGE\_INTEGER end\_time2;  // create and set timer  HANDLE tm1 = CreateWaitableTimer( NULL, false, NULL );  end\_time = -1 \* stop \* 10000000;  end\_time2.LowPart = (DWORD) (end\_time & 0xFFFFFFFF);  end\_time2.HighPart = (LONG) (end\_time >> 32);  SetWaitableTimer( tm1, &end\_time2, 0, NULL, NULL, false );  for( int i = 0; i < thrds; i++ ) {  params\* param = (params\*)malloc(sizeof(params));  param->num = i;  param->runflg = &runFlag;  HANDLE t1 = CreateThread( NULL, 0, Thread1, param, 0, &targetThreadId ); // create thread  CloseHandle( t1 );  }  GetSystemTime( &now ); // print current time  cout << "System Time " << now.wHour << " " << now.wMinute << " " << now.wSecond << endl;  WaitForSingleObject( tm1, INFINITE );  runFlag = false; // set flag CloseHandle( tm1 );  GetSystemTime( &now ); // print time after setting flag (ending)  cout << "System Time " << now.wHour << " " << now.wMinute << " " << now.wSecond << endl;  system( "pause" );  return 0;  }  DWORD WINAPI Thread1(LPVOID prm) {  while( 1 ) {  params arg = \*((params\*)prm);  Sleep( 1000 );  cout << arg.num << endl;  if( \*(arg.runflg) == false ) // check flag  break;  }  return 0;  } |

Результат выполнения:

*Рис.7: Результат выполнения task5.cpp*



В этой программе базовый поток ожидает сигнала от таймера (по истечению заданного времени), и после этого устанавливает флаг runFlag, который анализируют другие потоки, и если его значение равно false, то они заканчивают свое выполнение. Для контроля выполнения в начале и в конце программы выводится системное время.

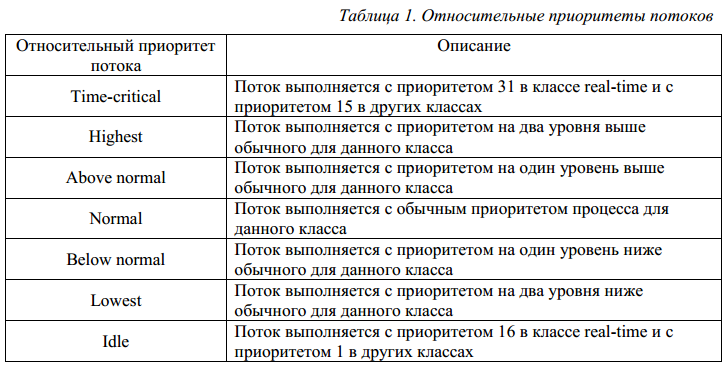
В аргументы программы было указано создание 4-ех потоков и длительность выполнения 5 секунд.

Использовать таймер с функциями WaitFor..Object можно не только в разных потоках, но и в разных процессах, т.е. ожидаемые таймеры могут служить средством межпроцессного взаимодействия.

# 3. Функции управления приоритетами процессов и потоков

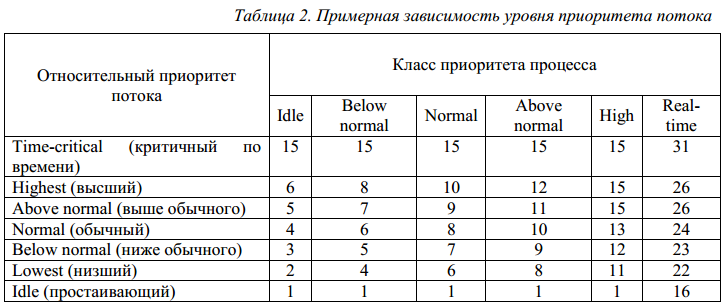
Windows поддерживает шесть классов приоритета процесса: idle (простаивающий), below normal (ниже обычного), normal (обычный), above normal (выше обычного), high (высокий), real-time (реального времени).

* Real-time - наивысший возможный приоритет. Потоки в этом процессе обязаны немедленно реагировать на события, их исполнение может привести к полной блокировке системы и требует осторожности в использовании этого класса.
* High – тоже потоки быстрого реагирования на события (этот класс присвоен Task Manager ). Above normal – класс приоритета промежуточный между normal и high, класс введенный в версии Windows 2000.
* Normal – потоки в этом процессе не предъявляют особых требований к выделению им процессорного времени.
* Below normal - класс приоритета промежуточный между normal и idle, класс введенный в Windows 2000.
* Idle – потоки в этом процессе выполняются, когда система не занята другой работой. Этот класс приоритета обычно используется для утилит, работающих в фоновом режиме.

Кроме того, Windows поддерживает семь относительных приоритетов потока: idle (простаивающий), lowest (низший), below normal (ниже обычного), normal (обычный), above normal (выше обычного), highest (высший) и time-critical (критичный по времени), описания которых указаны в таблице ниже.

Относительный приоритет потока принимает значение от 0 (самый низкий) до 31 (самый высокий), но программист работает не с численными значениями, а с так называемыми «константными». Это обеспечивает определенную гибкость и независимость при изменении алгоритмов планирования, а они меняются практически с каждой новой версией ОС, а с ними, соответственно, могут измениться и соотношения приоритетов.

Уровень приоритета формируется самой системой, исходя из класса приоритета процесса и относительного приоритета потока. Примерный вид зависимости уровня приоритета потока от класса приоритета процесса и от относительного приоритета потока представлен в таблице 2 (на примере Windows 2000).



Динамическое повышение приоритета предназначено для оптимизации общей пропускной способности и реактивности системы, при этом выигрывает не каждое приложение в отдельности, а система в целом. Windows может динамически повышать значение текущего приоритета потока в одном из следующих случаев:

1. после завершения операции ввода/вывода;
2. по окончании ожидания на каком-либо объекте исполнительной системы;
3. при нехватке процессорного времени и инверсии приоритетов.

Рассмотрим более подробно каждый из этих случаев.

1. После завершения операции ввода/вывода ОС временно динамически повышает приоритет потоков, предоставляя им больше шансов возобновить выполнение и обработать полученные данные. После динамического повышения приоритета поток в течение одного кванта выполняется с этим приоритетом. Следующий квант потоку выделяется с понижением приоритета на один уровень. Этот цикл продолжается до тех пор, пока приоритет не снизится до базового.
2. По окончании ожидания на каком-либо объекте исполнительной системы (например, SetEvent, ReleaseSemaphore) приоритет потока увеличивается на один уровень.
3. при инверсии приоритетов диспетчер настройки баланса просматривает очереди готовых потов и ищет потоки, которые находились в состоянии готовности (Ready) более 3 секунд. Обнаружив такой поток, диспетчер повышает его приоритет до 15 и выделяет ему квант вдвое больше обычного. По истечении двух квантов приоритет потока снижается до исходного уровня.

Система повышает приоритет только тех потоков, базовый приоритет которых попадает в область динамического приоритета (dynamic priority range), т.е. находится в пределах 1-15. ОС не допускает динамического повышения приоритета прикладного потока до уровней реального времени (выше 15).

Системные функции обслуживаются с приоритетами реального времени. ОС никогда не меняет приоритет потоков с уровнями реального времени (от 16 до 31). Это ограничение позволяет сохранять целостность системы и обеспечивает необходимый уровень безопасности.

Для работы с приоритетами используются следующие функции.

Функция SetThreadPriority() дает возможность установки базового уровня приоритета потока относительно класса приоритета его процесса.

|  |
| --- |
| BOOL SetThreadPriority(  HANDLE hThread, // дескриптор потока  int nPriority // уровень приоритета потока  ); |

Функция GetThreadPriorityBoost извлекает значение форсированного (динамически изменяемого) приоритета, который управляет состоянием заданного потока.

|  |
| --- |
| BOOL GetThreadPriorityBoost (  HANDLE hThread, // дескриптор потока  PBOOL pDisablePriorityBoost //состояние динамического изменения приоритета  ); |

Функция SetThreadPriorityBoost() разрешает/запрещает динамическое изменение приоритетов отдельного потока, не затрагивая остальные потоки.

|  |
| --- |
| BOOL SetThreadPriorityBoost (  HANDLE hThread, // дескриптор потока  BOOL DisablePriorityBoost // состояние форсирования приоритета  ); |

Когда поток запускается в одном из классов динамического приоритета, система временно повышает (форсирует) приоритет потока, чтобы вывести его из состояния ожидания. Если вызывается функция SetThreadPriorityBoost() с параметром DisablePriorityBoost=TRUE, приоритет потока не поднимается.

Программная установка флага priorityBoost позволяет для каждого из потоков отдельно указать возможность динамического повышения его приоритета. По умолчанию, boost потока и процесса «разрешено».

Рабочие потоки так же могут следить за своим приоритетом, при необходимости обновляя глобальную переменную priorityChange.

По умолчанию ОС разрешает динамическое изменение приоритетов для всего процесса (то есть для всех его потоков). Функция SetProcessPriorityBoost() оказывает влияние на все потоки указанного процесса, но не препятствует дальнейшему разрешению/запрещению динамического изменения приоритетов отдельных потоков.

Иногда для передачи управления другому потоку используют команду sleep(0), напомним, в этом случае нужно учитывать, что прогнозировать, какой из потоков запустится, довольно сложно, это осуществляется на усмотрение планировщика из очереди готовых в соответствии со сложившейся ситуацией в системе на текущий момент.

# 3. Программы для анализа влияния изменения классов приоритета процесса и принадлежащих ему потоков на выделение процессорного времени.

## 3.1.

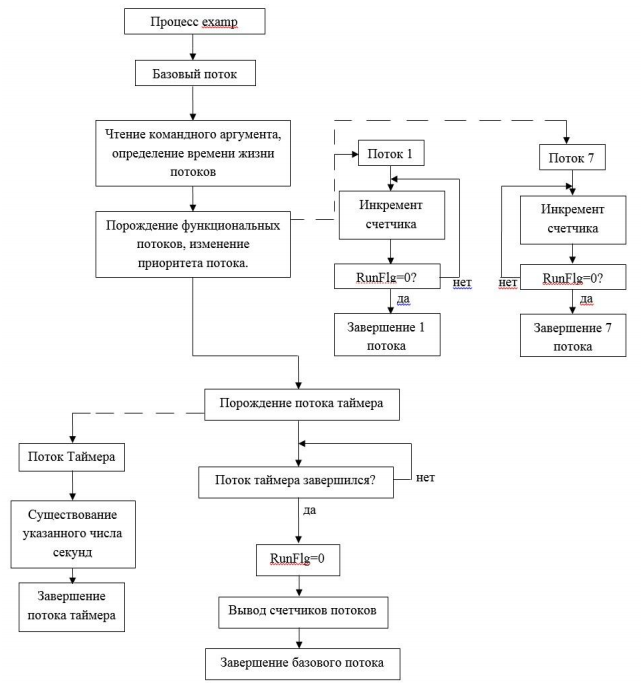
Подготовить программу, в которой у каждого из потоков свой приоритет отличный от других. Все они выполняют одинаковую работу, например, увеличивают каждый свой счетчик. Накопленное значение счетчика, таким образом, отражает относительное суммарное время выполнения потока.

Программа создает 7 потоков. Каждому из потоков присвоим уникальный приоритет. Все они выполняют одинаковую работу, увеличивают каждый свой счетчик. Накопленное значение счетчика, таким образом, отражает относительное суммарное время выполнения потока.

Предполагаем, так как приоритеты различны, то и время, отведенное на работу потокам различно (квант времени, выделяемый потокам, одинаков). Программа task6.cpp:

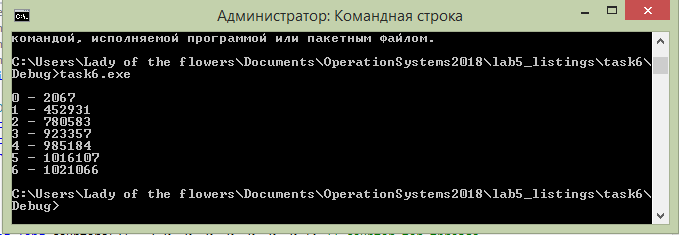
|  |
| --- |
| #define \_WIN32\_WINNT 0x0500  #include <stdio.h>  #include <windows.h>  #include <iostream>  using namespace std;  DWORD WINAPI Thread1( LPVOID );  int stop;  int sleep = 10000;  struct params {  int num;  bool\* runflg;  };  long long counters[7] = { 0, 0, 0, 0, 0, 0, 0 }; // counter for threads  int priority[7] = {THREAD\_PRIORITY\_IDLE,THREAD\_PRIORITY\_LOWEST,  THREAD\_PRIORITY\_BELOW\_NORMAL,THREAD\_PRIORITY\_NORMAL,  THREAD\_PRIORITY\_ABOVE\_NORMAL, THREAD\_PRIORITY\_HIGHEST,  THREAD\_PRIORITY\_TIME\_CRITICAL}; //massive of priorities in ascending order  int main( int argc, char\* argv[] ) { // on the command line set the thread life time  stop = 10;  DWORD targetThreadId;  bool runFlag = true; // initialization of the stream-timer structures  \_\_int64 end\_time;  LARGE\_INTEGER end\_time2;  HANDLE tm1 = CreateWaitableTimer( NULL, false, NULL ); // create timer  end\_time = -1 \* stop \* 10000000;  end\_time2.LowPart = (DWORD) (end\_time & 0xFFFFFFFF);  end\_time2.HighPart = (LONG) (end\_time >> 32);  SetWaitableTimer( tm1, &end\_time2, 0, NULL, NULL, false ); // start the timer  for( int i = 0; i < 7; i++ ) {  params\* param = (params\*)malloc(sizeof(params));  param->num = i;  param->runflg = &runFlag;  HANDLE t1 = CreateThread( NULL, 0, Thread1, param, 0, &targetThreadId ); // generation of a thread and  SetThreadPriority( t1, priority[i] ); // assigning priority to it  PBOOL ptr1 = (PBOOL)malloc(sizeof(BOOL));  GetThreadPriorityBoost( t1, ptr1 );  SetThreadPriorityBoost( t1, true ); // checking the dynamic allocation of priorities  CloseHandle( t1 ); // clean memory  }  WaitForSingleObject( tm1,INFINITE ); // waiting the thread of timer  runFlag = false; // ending working flag  CloseHandle( tm1 );  cout << endl;  for( int i = 0; i < 7; i++ ) {  cout << i << " - " << counters[i] << endl; // print results  }  return 0;  }  DWORD WINAPI Thread1(LPVOID prm) {  while( 1 ) {  DWORD WINAPI thrdid = GetCurrentThreadId(); // the value of the identifier of the calling thread  HANDLE WINAPI handle = OpenThread( THREAD\_QUERY\_INFORMATION , false, thrdid ); //descriptor of the thread  int WINAPI prio = GetThreadPriority(handle); // priority for the defined thread  params arg = \*((params\*)prm);  counters[arg.num]++;  if( prio != priority[arg.num] ) // displays when dynamic priority allocation is enabled  cout << endl << "Priority of " << arg.num << " is " << priority[arg.num] << " " << prio << " changed" << endl;  Sleep( 0 );  if( \*(arg.runflg) == false )  break;  }  return 1;  } |

*Рис.8: Схема выполнения программы (Структура программы)*



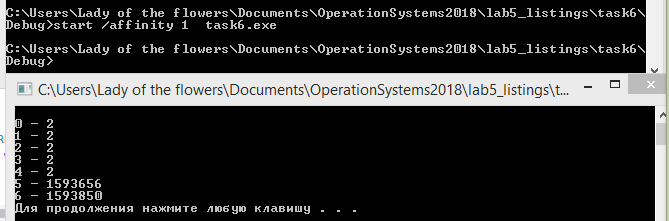
Результат выполнения:

*Рис.9: Результат выполнения task6.cpp на нескольких процессорах*



Дополнительно произведём запуск на одном процессоре:

*Рис.10: Результат выполнения task6.cpp на одном процессоре*



Как видно, при работе на одном процессоре, процессорный ресурс получают только потоки с двумя самыми высокими приоритетами. Остальные потоки выполняются только 1 раз, судя по счетчикам.

Слева - номер класса приоритета от низшего к высшему, а справа - значение счетчика, накопленное за все кванты, предоставленные потоку до истечения таймера. Очевидно, что потокам, у которых приоритет ниже, выделяется меньшее количество квантов времени для выполнения, и поэтому их счетчики соответственно меньше. При каждом запуске экспериментальные данные получаются различными, но пропорциональное соотношение между счетчиками потоков примерно сохраняется. Таким образом, пример демонстрирует, как ОС распределяет время процессора между потоками в зависимости от их приоритетов.

## 3.2. Усложним задачу и дополним ее возможностью управлять классом приоритетов процесса.

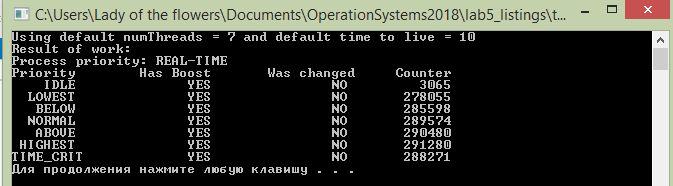
Код программы несколько изменим для получения более наглядного вывода результатов. Пусть программа по-прежнему создает 7 дополнительных потоков, со всеми возможными вариантами приоритета. Теперь в начале работы можно изменить класс приоритета процесса в целом. Каждый рабочий процесс выполняет увеличение связанного с ним счетчика (здесь типа int). После увеличения счетчика, поток отдает оставшуюся часть кванта времени остальным, с помощью вызова функции Sleep с параметром 0. Через заданное время рабочие потоки завершаются, а основной поток выводит результаты их работы в новом формате. Окончание работы происходит по сигналу от таймера. Заложена возможность задания произвольного количества потоков и времени жизни, отсчитываемого таймером.

|  |
| --- |
| #include <stdio.h>  #include <conio.h>  #include <windows.h>  #include <string.h>  #include <iostream>  using namespace std;  #define DEF\_THREADS 7  #define DEF\_TTL 10  DWORD WINAPI threadHandler(LPVOID);  HANDLE initTimer(int sec);  int getPriorityIndex(DWORD prClass);  int isFinish = 0;  long counters[7] = { 0, 0, 0, 0, 0, 0, 0 };  int priorities[7] = { THREAD\_PRIORITY\_IDLE, THREAD\_PRIORITY\_LOWEST, THREAD\_PRIORITY\_BELOW\_NORMAL, THREAD\_PRIORITY\_NORMAL,  THREAD\_PRIORITY\_ABOVE\_NORMAL, THREAD\_PRIORITY\_HIGHEST, THREAD\_PRIORITY\_TIME\_CRITICAL };  char charPrio[7][10] = { "IDLE", "LOWEST", "BELOW", "NORMAL", "ABOVE", "HIGHEST", "TIME\_CRIT" };  char charProcPrio[6][10] = { "IDLE", "BELOW", "NORMAL", "ABOVE", "HIGH", "REAL-TIME" };  int procPriorities[6] = { IDLE\_PRIORITY\_CLASS, BELOW\_NORMAL\_PRIORITY\_CLASS, NORMAL\_PRIORITY\_CLASS,  ABOVE\_NORMAL\_PRIORITY\_CLASS, HIGH\_PRIORITY\_CLASS, REALTIME\_PRIORITY\_CLASS };  int priorityBoost[7] = { 0, 0, 0, 0, 0, 0, 0 };  int priorityChange[7] = { 0, 0, 0, 0, 0, 0, 0 };  int main( int argc, char\* argv[] ) {  int numThreads = DEF\_THREADS;  int threadLive = DEF\_TTL;  if( argc < 2 )  cout << "Using default numThreads = " << numThreads << " and default time to live = " << threadLive << endl;  else if( argc < 3 )  cout << "Using default time to live = " << threadLive << endl;  else {  numThreads = atoi( argv[1] );  threadLive = atoi( argv[2] );  if( numThreads <= 0 || threadLive <= 0 ) {  cout << "All arguments must be numbers!!!!" << endl;  exit(0);  }  }  HANDLE t = initTimer( threadLive );  HANDLE t1;  // set the priority of the process = IDLE (or other)  SetPriorityClass( GetCurrentProcess(), REALTIME\_PRIORITY\_CLASS );  for( int i = 0; i < numThreads; i++ ) {  t1 = CreateThread( NULL, 0, threadHandler, (LPVOID)i, 0, NULL );  SetThreadPriority( t1, priorities[i] );  SetThreadPriorityBoost( t1, true );  GetThreadPriorityBoost( t1, &priorityBoost[i] );  CloseHandle( t1 );  }  WaitForSingleObject( t, INFINITE ); // Wait all threads - comment-uncommitted line depending on the task  CloseHandle( t );  isFinish = 1;  char hasBoost[4];  char wasChanged[4];  int priorIdx = getPriorityIndex( GetPriorityClass( GetCurrentProcess() ) );  cout << "Result of work:" << endl;  cout << "Process priority: " << charProcPrio[priorIdx] << endl;  cout << "Priority\tHas Boost\tWas changed\tCounter" << endl;  for( int i = 0; i < 7; i++ ) {  priorityBoost[i] == 0 ? strcpy( hasBoost, "NO" ) : strcpy( hasBoost, "YES" );  priorityChange[i] == 0 ? strcpy( wasChanged, "NO") : strcpy( wasChanged, "YES" );  printf( "%8s\t%9s\t%10s\t%7d\n", charPrio[i], hasBoost, wasChanged, counters[i] );  }  system( "pause" );  return 0;  }  DWORD WINAPI threadHandler( LPVOID prm ) {  int myNum = (int)prm;  int priority = 0;  for( ;; ) {  ++counters[myNum];  priority = GetThreadPriority( GetCurrentThread() );  if( priority != priorities[myNum] )  priorityChange[myNum] = 1;  if( isFinish )  break;  Sleep( 0 );  }  return 0;  }  HANDLE initTimer( int sec ) {  \_\_int64 end\_time;  LARGE\_INTEGER end\_time2;  HANDLE tm = CreateWaitableTimer( NULL, false, "timer" );  end\_time = -1 \* sec \* 1000000;  end\_time2.LowPart = (DWORD)(end\_time & 0xFFFFFFFF);  end\_time2.HighPart = (LONG)(end\_time >> 32);  SetWaitableTimer(tm, &end\_time2, 0, NULL, NULL, false);  return tm;  }  int getPriorityIndex( DWORD prClass ) {  for( int i = 0; i < 6; ++i ) {  if( procPriorities[i] == prClass )  return i;  }  return 0;  } |

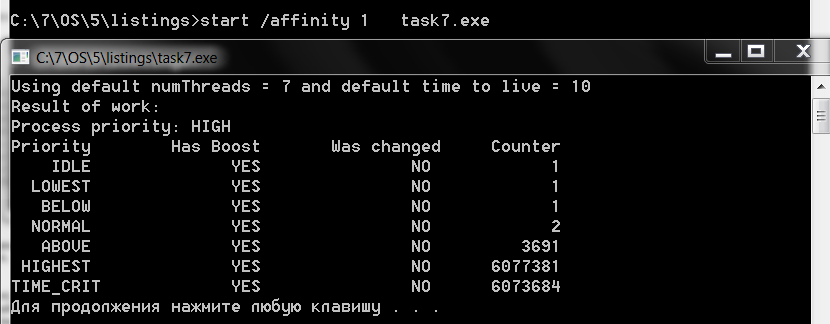
Результаты выполнения:

В первом случае почти все потоки (кроме потоков с самым низким приоритетом) получили достаточное количество процессорного времени.

*Рис.11: Результат выполнения task7.cpp на нескольких процессорах*



*Рис.12: Результат выполнения task7.cpp на одном процессоре*



Как видно из результатов во втором случае на одном процессоре, из-за включенной возможности динамического изменения приоритетов операционной системой, были изменены первые 3 приоритета, что повлияло на результаты.

## 3.3. Задача. Анализ поведения системных функций динамического управления приоритетами процессов и потоков.

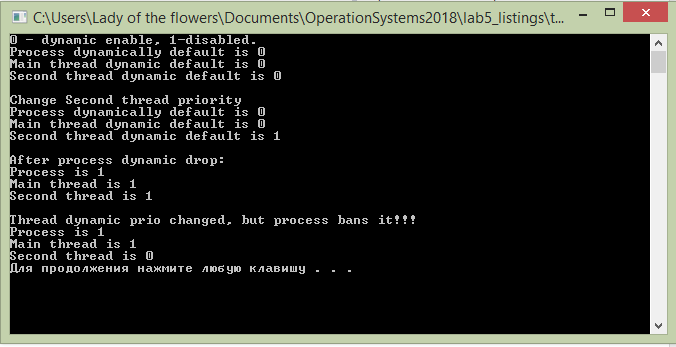
С помощью программы определим, назначается ли динамическое изменение приоритетов по умолчанию, на все ли потоки воздействует функция SetProcessPriorityBoost(), возможно ли разрешение отдельному потоку в процессе динамически изменять приоритет, если для процесса это запрещено.

Программа task8.cpp:

|  |
| --- |
| #include <stdio.h>  #include <iostream>  #include <string>  #include <windows.h>  using namespace std;  void thread() {  while( true ) {  Sleep(2000);  }  }  int main( int argc, char \*argv[] ) {  BOOL dynamic;  HANDLE processHandle, mainThread, secondThread; // threads  DWORD secondID; processHandle = GetCurrentProcess();// descriptor of the current process  mainThread = GetCurrentThread(); // descriptor of the current thread  //create a second thread  secondThread = CreateThread(NULL,0, (LPTHREAD\_START\_ROUTINE) thread,NULL,NULL,&secondID); // create thread 2  //print default values  cout << "0 - dynamic enable, 1-disabled." << endl;  GetProcessPriorityBoost( processHandle,&dynamic ); // value of dynamic process priority  cout << "Process dynamically default is " << dynamic << endl;  GetThreadPriorityBoost( mainThread,&dynamic );  cout << "Main thread dynamic default is " << dynamic << endl;  GetThreadPriorityBoost( secondThread,&dynamic );  cout << "Second thread dynamic default is " << dynamic << endl << endl;  cout << "Change Second thread priority" << endl;  if( !SetThreadPriorityBoost( secondThread, true )){ // increase priority of thread 2  cout << "Error on thread change!" << endl;  return 2;  }  GetProcessPriorityBoost( processHandle,&dynamic );  cout << "Process dynamically default is " << dynamic << endl;  GetThreadPriorityBoost( mainThread, &dynamic );  cout << "Main thread dynamic default is " << dynamic << endl;  GetThreadPriorityBoost( secondThread, &dynamic );  cout << "Second thread dynamic default is " << dynamic << endl << endl;  //change process dynamically  if( !SetProcessPriorityBoost( processHandle, true ) ) { // increase priority of process  cout << "Error on prir change!" << endl;  return 1;  }  cout << "After process dynamic drop:" << endl;  GetProcessPriorityBoost( processHandle, &dynamic );  cout << "Process is " << dynamic << endl;  GetThreadPriorityBoost( mainThread, &dynamic );  cout << "Main thread is " << dynamic << endl;  GetThreadPriorityBoost( secondThread, &dynamic );  cout << "Second thread is " << dynamic << endl << endl;  //may be can change thread dynamic prio?  if( !SetThreadPriorityBoost( secondThread, false ) ){ // lower priority of thread 2  cout << "We cannot change if process ban dynamic prio!!!" << endl;  return 3;  } else {  cout << "Thread dynamic prio changed, but process bans it!!!" << endl;  GetProcessPriorityBoost( processHandle, &dynamic );  cout << "Process is " << dynamic << endl;  GetThreadPriorityBoost( mainThread, &dynamic );  cout << "Main thread is " << dynamic << endl;  GetThreadPriorityBoost( secondThread, &dynamic );  cout << "Second thread is " << dynamic << endl;  }  return 0;  } |

Результаты выполнения:

*Рис.13: Результат выполнения task8.cpp*



Из листингов видно, что по умолчанию для процессов и потоков динамическое изменение приоритетов разрешено. Потом для второго потока было запрещено изменение динамического приоритета с помощью функции SetThreadPriorityBoost. Если запретить динамическое изменение приоритетов для процесса (SetProcessPriorityBoost), то изменение также будет запрещено и для всех потоков. Разрешение отдельному потоку в процессе динамически изменять приоритет возможно, если для процесса это запрещено.

# 4. Самостоятельные задания

## 1. Исследуйте результаты работы программы 3.1 и 3.2 в зависимости от того, какой приоритет назначается базовому потоку: аbove\_normal, idle\_priority class, high priority class, normal priority class и др.; своими экспериментальными данными заполните таблицу 3, приведенную ниже, с точным указанием для какой ОС и на каком отладочном комплексе проводились измерения

Исследование проводилось на ОС Windows 7.

*Таблица 3: Исследование программы task7.cpp для различных значений приоритета базового потока*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Относительный приоритет потока | Класс приоритета процесса | | | | | |
| Idle | Below normal | Normal | Above Normal | High | Real-time |
| THREAD\_PRIORITY\_IDLE | 288780 | 1 | 1088618 | 350736 | 525518 | 510136 |
| THREAD\_PRIORITY\_LOWEST | 1223304 | 681830 | 32922 | 387989 | 535506 | 532252 |
| THREAD\_PRIORITY\_BELOW\_NORMAL | 493116 | 2662163 | 859527 | 1146211 | 540670 | 538993 |
| THREAD\_PRIORITY\_NORMAL | 1279345 | 467059 | 509803 | 727718 | 526166 | 532407 |
| THREAD\_PRIORITY\_ABOVE\_NORMAL | 403863 | 1075643 | 474074 | 443345 | 1479907 | 510344 |
| THREAD\_PRIORITY\_HIGHEST | 507184 | 1375310 | 515203 | 491901 | 535495 | 1449402 |
| THREAD\_PRIORITY\_TIME\_CRITICAL | 502473 | 1025502 | 1357760 | 502891 | 540714 | 539043 |

Без ожидания завершения всех потоков:

*Таблица 4: Исследование программы task7.cpp для различных значений приоритета базового потока*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Относительный приоритет потока | Класс приоритета процесса | | | | | |
| Idle | Below normal | Normal | Above Normal | High | Real-time |
| THREAD\_PRIORITY\_IDLE | 1 | 314058 | 117 | 0 | 0 | 159 |
| THREAD\_PRIORITY\_LOWEST | 1 | 1 | 1 | 0 | 1 | 116 |
| THREAD\_PRIORITY\_BELOW\_NORMAL | 1 | 498005 | 82 | 1505 | 1 | 1 |
| THREAD\_PRIORITY\_NORMAL | 1 | 809637 | 3 | 1499 | 1 | 1 |
| THREAD\_PRIORITY\_ABOVE\_NORMAL | 1 | 314144 | 3 | 3 | 1 | 1 |
| THREAD\_PRIORITY\_HIGHEST | 1 | 19 | 2 | 1 | 1 | 1 |
| THREAD\_PRIORITY\_TIME\_CRITICAL | 1 | 395471 | 1 | 1 | 1 | 1 |

Судя по результатам эксперимента, класс приоритета процесса влияет на процессорное время, отводимое каждому процессу, но не так, как мы ожидали: фоновый режим получает больше процессорного времени, нежели процесс с классом приоритета реального времени.

## 2. Модифицируйте программу 3.2 для заполнения таблицы 2 текущими данными вашего эксперимента. Сделайте выводы.

*Таблица 5: Исследование модифицированной программы task7.cpp для различных значений приоритета базового потока*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Относительный приоритет потока | Класс приоритета процесса | | | | | |
| Idle | Below normal | Normal | Above Normal | High | Real-time |
| THREAD\_PRIORITY\_IDLE | 2110679 | 2110680 | 2110681 | 2110682 | 2110683 | 2110684 |
| THREAD\_PRIORITY\_LOWEST | 2101362 | 2101363 | 2101364 | 2101365 | 2101366 | 2101367 |
| THREAD\_PRIORITY\_BELOW\_NORMAL | 2116581 | 2116582 | 2116583 | 2116584 | 2116585 | 2116586 |
| THREAD\_PRIORITY\_NORMAL | 2141917 | 2141918 | 2141919 | 2141920 | 2141921 | 2141922 |
| THREAD\_PRIORITY\_ABOVE\_NORMAL | 2134097 | 2134098 | 2134099 | 2134100 | 2134101 | 2134102 |
| THREAD\_PRIORITY\_HIGHEST | 2130892 | 2130893 | 2130894 | 2130895 | 2130896 | 2130897 |
| THREAD\_PRIORITY\_TIME\_CRITICAL | 2158187 | 2158188 | 2158189 | 2158190 | 2158191 | 2158192 |

Без ожидания завершения всех потоков:

*Таблица 6: Исследование модифицированной программы task7.cpp для различных значений приоритета базового потока*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Относительный приоритет потока | Класс приоритета процесса | | | | | |
| Idle | Below normal | Normal | Above Normal | High | Real-time |
| THREAD\_PRIORITY\_IDLE | 169 | 170 | 171 | 172 | 173 | 174 |
| THREAD\_PRIORITY\_LOWEST | 104 | 105 | 106 | 107 | 108 | 109 |
| THREAD\_PRIORITY\_BELOW\_NORMAL | 68 | 69 | 70 | 71 | 72 | 73 |
| THREAD\_PRIORITY\_NORMAL | 25 | 26 | 27 | 28 | 29 | 30 |
| THREAD\_PRIORITY\_ABOVE\_NORMAL | 15 | 16 | 17 | 18 | 19 | 20 |
| THREAD\_PRIORITY\_HIGHEST | 6 | 7 | 8 | 9 | 10 | 11 |
| THREAD\_PRIORITY\_TIME\_CRITICAL | 1 | 2 | 3 | 4 | 5 | 6 |

По результатам исследования можно предположить, что в данной системе приоритет потока не зависит от класса потока.

## 3. С помощью утилит CPU Stress, позволяющих нагружать систему, и утилиты мониторинга ProcessExplorer() (или иных утилит) зафиксируйте динамическое изменение приоритетов, приведите результаты в отчете.

Используем утилиту CPU Stress для запуска процесса с возможность динамического изменения приоритета. Для просмотра таблицы процессов с дополнительными данными используем утилиту ProcessExplorer, Устанавливая один из трех возможных в данной программе классов приоритета для процесса, увидим изменение значения колонки Priority в таблице процессов утилиты ProcessExplorer.

*Рис. 14: До выполнения процесса с приоритетом BELOW\_NORMAL\_PRIORITY\_CLASS*



*Рис. 15: Во время выполнения процесса с приоритетом BELOW\_NORMAL\_PRIORITY\_CLASS*



*Рис. 16: До выполнения процесса с приоритетом REALTIME\_PRIORITY\_CLASS*



*Рис. 17: Во время выполнения процесса с приоритетом REALTIME\_PRIORITY\_CLASS*



## 4. Создайте программу, демонстрирующую возможность наследования.

1. дескриптора порождающего процесса,
2. дескрипторов открытых файлов,

для выполнения этого задания следует учесть, что по умолчанию наследование в Windows отключено и для возможности наследования, необходимо:

1. разрешить процессу-потомку наследовать дескрипторы,
2. сделать дескрипторы наследуемыми

Программа father.cpp:

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include <stdio.h>  #include <windows.h>  #include <assert.h>  #include <stdlib.h>  int main(void){  TCHAR cmdLine[100];  STARTUPINFO si;  SECURITY\_ATTRIBUTES sa;  PROCESS\_INFORMATION pi;  HANDLE hFile;  char DataBuffer[] = "Father proc!\n";  DWORD dwBytesToWrite = (DWORD)strlen(DataBuffer);  DWORD dwBytesWritten = 0;  printf("\n");  system("pause");  sa.nLength = sizeof(SECURITY\_ATTRIBUTES);  sa.lpSecurityDescriptor = NULL;  sa.bInheritHandle = TRUE;  hFile = CreateFile(TEXT("file.txt"), // name of the write  GENERIC\_WRITE, // open for writing  0, // do not share  $\&$sa, // default security  CREATE\_NEW, // create new file only  FILE\_ATTRIBUTE\_NORMAL, // normal file  NULL); // no attr. template  if (hFile == INVALID\_HANDLE\_VALUE)  {  printf("Error #%d", GetLastError());  return -1;  }  BOOL bErrorFlag = WriteFile(  hFile, // open file handle  DataBuffer, // start of data to write  dwBytesToWrite, // number of bytes to write  $\&$dwBytesWritten, // number of bytes that were written  NULL); // no overlapped structure  if (FALSE == bErrorFlag)  {  printf("Terminal failure: Unable to write to file. Error #%d\n", GetLastError());  }  wsprintf(cmdLine, TEXT("son.exe %d"), (int)hFile);  ZeroMemory($\&$si, sizeof(STARTUPINFO));  si.cb = sizeof(STARTUPINFO);  if (!CreateProcess(NULL,cmdLine,NULL,NULL,TRUE,CREATE\_NEW\_CONSOLE,NULL, NULL,$\&$si,$\&$pi)){  printf("Error creating process: %d\n", GetLastError());  system("pause");  return GetLastError();  }  CloseHandle(pi.hThread);  CloseHandle(pi.hProcess);  system("pause");  CloseHandle(hFile);  return 0;  } |

Программа son.cpp:

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include <stdio.h>  #include <windows.h>  #include <assert.h>  #include <stdlib.h>  int main(int argc, char \*argv[]){  HANDLE hFile;  char DataBuffer[] = "Son proc\n";  DWORD dwBytesToWrite = (DWORD)strlen(DataBuffer);  DWORD dwBytesWritten = 0;  hFile = (HANDLE)atoi(argv[1]);  BOOL bErrorFlag = WriteFile(  hFile, // open file handle  DataBuffer, // start of data to write  dwBytesToWrite, // number of bytes to write  $\&$dwBytesWritten, // number of bytes that were written  NULL); // no overlapped structure  if (FALSE == bErrorFlag)  {  printf("Terminal failure: Unable to write to file. Error #%d\n", GetLastError());  }  if (CloseHandle(hFile) == 0)  printf("Error %d\n", GetLastError());  //system("pause");  return 0;  } |

Содержимое файла, созданного в текущей директории представлено ниже. Наличие вывода обеих программ говорит об успешной передаче дескрипторов открытых файлов.

Содержимое файла test.txt:

|  |
| --- |
| Father proc!  Son proc! |

Таким образом, в операционных системах семейства Windows, в отличие от операционных систем на основе Unix, при порождении нового процесса файловые и иные дескрипторы не наследуются по умолчанию. Для осуществления наследования необходимо явно указать факт наследования и то, какие псевдоуказатели наследовать.

В операционных системах Unix наоборот: файловые и иные дескрипторы наследуются по умолчанию, необходимо помечать те дескрипторы, которые не нужно наследовать.

# Вывод

В данной работе было изучено управление процессами и потоками в ОС Windows.

Используя функцию CreateThread, можно создавать новые потоки в рамках одного процесса, тем самым распараллеливая вычисления. Как и функция CreateProcess, функция CreateThread имеет большое количество параметров, предоставляя широкие возможности по созданию потоков.

Такие системные объекты, как таймеры ожидания, могут быть использованы для синхронизации потоков в ОС семейства Windows.

Windows поддерживает шесть классов приоритета процесса, а так же 7 возможных приоритетов потока в рамках класса приоритета процесса. В ОС существует динамическое повышение приоритета, предназначенное для оптимизации общей пропускной способности системы: операционная система может автоматически повышать значение текущего приоритета после завершения операции ввода/вывода, по окончанию ожидания на каком-либо системном объекте, при нехватке процессорного времени.

Потокам, у которых приоритет ниже, выделяется меньшее количество квантов времени для выполнения. Примеры, рассмотренные в работе, демонстрируют, как ОС распределяет время процессора между потоками в зависимости от их приоритетов. Аналогично система поступает с процессами и их классами приоритетов.

# Список литературы

1. Душутина Е.В. Межпроцессные взаимодействия в операционных системах – СПб, 2014 г, 136 с.